UNDERFLOW REVISITED

Annie Cuyt*
Peter Kuterna•
Brigitte Verdonk°
Dennis Verschaeren†

Department of mathematics and computer science
Universiteit Antwerpen (UIA), Universiteitsplein 1
B–2610 Wilrijk, Belgium
email: {cuyt, kuterna, verdonk, vschaer}@uia.ua.ac.be

Keywords

Computer arithmetic, IEEE floating-point, underflow exception

Abstract

Underflow is a floating-point phenomenon. Although the use of gradual underflow as defended in [2] and [5] is now widespread, most numerical analysts may not be aware of the fact that several implementations of the same principle are in existence, leading to different behaviour of code on different platforms, mainly with respect to exception signaling. We intend to thoroughly discuss the slight differences among these implementations. Examples will be taken from current hardware and from our own multiprecision software class library. Throughout the discussion the focus is on the analysis of the phenomenon and not on any implementation issues. Many programmers are also unaware of the fact that the IEEE 754 and 854 standards do not guarantee that a program will deliver identical results on all conforming systems. Of all the differences that can occur cross-platform, the underflow exception is just one.

* Research Director FWO-Vlaanderen
• Supported by an NOI-grant from the Universiteit Antwerpen (UIA)
° Postdoctoral Fellow FWO-Vlaanderen
† Research Fellow IWT (Institute for Science and Technology)

1. Underflow and IEEE 754-854

Let us denote by $I\!F(\beta, t, L, U)$ the set of normalized floating-point numbers $\pm d_0.d_1 \ldots d_{t-1} \times \beta^e$ where

$$0 \leq d_i \leq \beta - 1$$
$$d_0 \neq 0$$
$$L \leq e \leq U$$

In order to be able to take care of overflow, the set is enlarged with the representation $\pm 1.0 \ldots 0 \times \beta^{U+1}$ for signed infinity. In order to take care of underflow the set is also enlarged with the denormalized numbers $\pm 0.d_1 \ldots d_{t-1} \times \beta^L$, where $\pm 0.0 \ldots 0 \times \beta^L$ represents signed zero. We speak of denormalized numbers rather than unnormalized, because they cannot be normalized with the exponent range being bounded by $[L, U]$. The introduction of denormal numbers enables the implementation of the primary underflow mechanism of gradual underflow, which roughly speaking rounds tiny floating-point numbers less than β^L in magnitude to denormal numbers, rather than to zero. The internal representation of these denormal numbers and signed zeroes is with exponent $L - 1$, flagging that the leading digit $d_0 = 0$ while the actual exponent $e = L$.

Because the set of floating-point numbers is a discrete approximation of the real number set, each arithmetic operation introduces a rounding error. In round to nearest, the default rounding mode, the relative error made when approximating $x*y$ by its nearest floating-point number $\bigcirc(x*y)$, where $* \in \{+, -, \times, /\}$, is at most

$$\left| \frac{x * y - \bigcirc(x * y)}{x * y} \right| \leq \frac{\beta}{2}\beta^{-t} \qquad x, y \in I\!F(\beta, t, L, U) \tag{1}$$

unless $x * y$ overflows or underflows. Implementing $x * y$ such that its machine version $x \circledast y$ delivers $\bigcirc(x * y)$ is called an exactly rounded implementation. When taking gradual underflow into account the error analysis has to be reformulated as

$$\bigcirc(x * y) = (x * y)(1 + \epsilon) + \eta \qquad |\epsilon| \leq \frac{\beta}{2}\beta^{-t} \qquad |\eta| \leq (\frac{\beta}{2}\beta^{-t})\beta^L \qquad \epsilon\eta = 0 \tag{2}$$

unless $x * y$ overflows. Here ϵ expresses the relative error that occurs when $|\bigcirc(x * y)|$ is larger than or equal to the underflow threshold β^L. For denormalized results, η expresses the absolute error which is at most $\frac{\beta}{2}\beta^{-t}\beta^L$ [5]. In round up, round down or trunc the relative bound of $(\frac{\beta}{2}\beta^{-t})$ doubles to β^{-t+1} in (1) and (2). We also have to point out that at most one of ϵ and η is nonzero, in other words $\epsilon\eta = 0$, and that moreover always $\eta = 0$ when the arithmetic operation is either addition or subtraction [9]. It is easy to prove that the use of denormalized numbers allows to represent all tiny results of the floating-point addition or subtraction operation without rounding error, in other words exactly.

The underflow exception flag was introduced to signal to the user of a programming environment that (1) is no longer valid and that $\eta \neq 0$ has occurred in (2). The IEEE standards [6, 8] relax this condition in the sense that the underflow exception should be signaled "at least" when $\eta \neq 0$. A commentary to the standard however [1] encourages the stricter criterion for setting the underflow flag. That is, it should be set whenever the delivered result is different from what would be delivered in a system with the same precision but with an unbounded (or large enough) exponent range. But it is the rule rather than the exception that many more underflow alarms go off.

The IEEE standards specify that underflow (in non-trapping mode) should be signaled at the occurrence of the following two correlated events. One is the creation of a tiny nonzero result in the interval $]-\beta^L, +\beta^L[$. The other is the loss of accuracy during the approximation of this tiny result, usually

by a denormalized number. Tininess may be detected either after rounding or before rounding. Loss of accuracy may be detected as either a case of inexactness (the delivered result differs from the result computed with unbounded precision and unbounded exponent range) or a case of denormalization loss (the denormalized result differs from the result delivered with unbounded exponent range). Since the decision to denormalize is taken after rounding, only the following combinations of tininess and loss of accuracy can occur: tiny before rounding and inexact, tiny after rounding and inexact, tiny after rounding and denormalization loss. Let us now discuss these three possible ways to detect the underflow exception in accordance with the IEEE standards. Although several options are possible for the implementation of the underflow exception, the IEEE standard requires that underflow be detected in the same way for all operations in a single programming environment.

We introduce the notations `result_ext` for the exact result of the arithmetic operation $x * y$ (unbounded precision and unbounded exponent range) and `result_tmp` for the normalized, rounded result (to t β-digits) of $x * y$ with unbounded exponent range. We let `result` denote the (possibly denormalized) delivered floating-point result. Then the following three slightly different implementations are consistent with the specifications of the IEEE standards for the underflow exception:

(W) `result_ext` is tiny (this is before rounding) and cannot be delivered exactly to `result`, in other words

$$|\texttt{result\_ext}| < \beta^L$$
$$\texttt{result} \neq \texttt{result\_ext}$$

(V) `result_tmp` is tiny (this is after rounding to t β-digits) and different from `result_ext` or `result` or both, in other words

$$|\texttt{result\_tmp}| < \beta^L$$
$$\texttt{result} \neq \texttt{result\_ext}$$

(U) `result_tmp` is tiny, possibly different from `result_ext` and has to be denormalized just like in the previous case; but what's worse, in the process of the denormalization, nonzero trailing β-digits are lost, implying the condition $\eta \neq 0$ in (2),

$$|\texttt{result\_tmp}| < \beta^L$$
$$\texttt{result} \neq \texttt{result\_tmp}$$

It is clear that if condition U is satisfied, also condition V is satisfied, because a denormalization loss implies inexactness. Moreover if condition V is true also condition W is true, because tininess after rounding implies tininess before rounding. The situation where case U does not occur while `result_tmp` is tiny, is in fact not so alarming because here the tiny result can be denormalized without losing any nonzero β-digits. For instance, with $t = 3$ and $\beta = 2$, the conditions for U-underflow are not satisfied in case of

$$\begin{aligned} \texttt{result\_ext} &= 1.0111 \times 2^{L-1} \\ \texttt{result\_tmp} &= 1.10 \times 2^{L-1} \\ \texttt{result} &= 0.11 \times 2^{L} \end{aligned} \tag{3}$$

whereas they are in case of

$$\begin{aligned} \texttt{result\_ext} &= 1.0111 \times 2^{L-2} \\ \texttt{result\_tmp} &= 1.10 \times 2^{L-2} \\ \texttt{result} &= 0.01 \times 2^{L} \end{aligned} \tag{4}$$

The difference between the two cases lies in the fact that in (3) $\eta = 0$ whereas in (4) $\eta \neq 0$, the first case obeying (1) and the second case not. The difficulty in detecting pure U-underflow stems from the inexact flag which in most implementations does not allow to distinguish between `result_ext` $\neq$ `result_tmp` and `result_tmp` $\neq$ `result`. Observe that the inexact exception can be raised at several occasions during the computation of $x \circledast y$: the significand of the normalized `result_ext` may contain more than t bits or, while both the significands of `result_ext` and `result_tmp` contained at most t bits, some trailing nonzero bits may be lost at the very end during denormalization.

In addition to signaling underflow at the occurrence of a U case, a V implementation also signals underflow when `result_tmp` is tiny but does not suffer a denormalization loss: the inexact condition arises during rounding but (1) is not violated. In addition to signaling underflow in the U and V cases, a W implementation even signals underflow when the delivered floating-point result, though inexact, isn't tiny anymore: the delivered result equals $\pm\beta^L$ after rounding. Since all potentially dangerous underflow cases that complicate the error analysis should be flagged, the U implementation is the minimal implementation required. However, owing to the difficulty of implementing this scheme, the IEEE standard allows setting the underflow flag whenever the unrounded or final result is tiny and the infinitely precise result cannot be delivered exactly.

2. Underflow signaling in some implementations

We discuss a W and a V hardware implementation, respectively by SUN for their UltraSparc processors and by INTEL for their Pentium processors. At the end we also mention a multiprecision C++ class library developed by one of the authors, which supports a U implementation. The consistency of each implementation was tested using a large set of test vectors which is a generalization of Coonen's [3] and Hough's [7] sets of test vectors and which will be publicly available soon [4]. For each of the precisions (single, double, 64 bit extended on INTEL and 113 bit quadruple on SUN) our test set for multiplication contained 1152 cases of U underflow, an extra 176 cases of V$\wedge\neg$U underflow and an additional 64 cases of W$\wedge\neg$V underflow. Analogously the test set for division contained 286 cases of alarming U underflow and 51 cases of V$\wedge\neg$U underflow. One can show [4] that W$\wedge\neg$V underflow cannot occur during division. It was already pointed out in the previous section that underflow does not occur in addition and subtraction when gradual underflow is supported.

2.1 SUN UltraSparc

Although SUN implemented V underflow signaling in their single and double precision SuperSparc processors, they have switched to a W implementation in their single and double precision UltraSparc. Both the V and the W implementations are easier and faster to handle than the detection of a pure U case. The implementation on the UltraSparc signals all W, V and U cases included in our test set, as is required for a proper W implementation, since U implies V, which in its turn implies W. Let us give an example of a test vector that signals underflow on the UltraSparc and not on the SuperSparc.

If we multiply $x = 0.11\ldots1 \times 2^L$ by $y = 1.00\ldots01 \times 2^0$ in any of the available precisions, then

$$\begin{aligned} \texttt{result\_ext} &= 0.11\ldots1\ldots111 \times 2^L \qquad 2t-1 \text{ bits} \\ &= 1.1\ldots1\ldots111 \times 2^{L-1} \end{aligned}$$

When rounding the tiny `result_ext` to nearest or upward, one obtains `result_tmp` $= 1.0 \times 2^L$ or the smallest normalized float. It is clear that the conditions for W underflow are satisfied while these for V underflow are not.

2.2 INTEL PC family

The INTEL processors are extended-based. The default working precision is $t = 64$ and the default exponent range $[L, U] = [-16382, 16383]$. Single or double precision arithmetic can be mimiced by changing the precision control (often also referred to as rounding precision) to respectively 24 or 53. But this leads to a change in the underflow behaviour (as described in section 2.2.1) and to erroneous double rounding in some cases (as described in section 2.2.2).

2.2.1 Underflow strategy and precision control

In its extended precision the INTEL Pentium implements the V underflow strategy. However, it behaves like a U implementation in single and double precision, as a result of the fact that the hardware is extended-based: only the precision of single ($t = 24$) or double ($t = 53$) is mimiced but the exponent range is still that of the extended precision (15 bits wide). Hence single or double precision tiny results are only recognized as tiny when stored from the extended precision register to memory. At that moment also the single or double precision denormalization takes place, resulting in U underflow detection.

Let us elaborate this in a some more detail. Assume that the precision control is set to double ($t = 53$) and consider for instance the two double precision operands $a = b = 1.0 \times 2^{-1022}$, the smallest normal number in double precision. Now consider the C program statements

```
double a = b;
long double c = a * b;
double d = a * b;
```

Most numerical analysts will agree with the idea that the second statement consists of two operations, namely a double precision multiplication and a conversion of the result to extended precision, and that the last statement is essentially one single operation, namely the double precision product $a \times b$ copied to the double precision variable d. So we expect to get

$$\begin{aligned} a \times b &= \text{double}(1.0 \times 2^{-2044}) = 0 \qquad \text{underflow, inexact} \\ c &= 0 \\ d &= 0 \end{aligned}$$

This is not the case on an INTEL platform. Because the hardware is extended-based, the second statement actually behaves like most of us think the last statement does: a and b are copied to the extended precision registers of the INTEL FPU, the product is carried out in the extended precision register and rounded according to the precision of precision control (in this case $t = 53$) but with extended precision exponent range. The result is then copied to the extended precision variable c. In the same way, the last statement is essentially a compound statement, what many don't realize: a and b are copied to the extended precision registers of the INTEL FPU, the product is again carried out in the extended precision register and rounded to the precision of precision control ($t = 53$) with extended precision exponent range. The result is then converted to the double precision variable d. This conversion to double precision memory implies a reduction in the range of representable exponents (from $[-16382, 16383]$ to $[-1022, 1023]$) and hence we get

$$\begin{aligned} a \times b &= 1.0 \times 2^{-2044} \\ c &= 1.0 \times 2^{-2044} \\ d &= 0 \qquad \text{underflow, inexact} \end{aligned}$$

Even though the precision control is set to $t = 53$, the product $a \times b$ does not underflow because the extended precision exponent range is large enough. Underflow and inexactness, which is due to denormalization loss, are only detected when the result of the product is stored in d.

So for the evaluation of expressions with the precision control set to 24 or 53 bits, the INTEL actually uses a hybrid format with 15 bits for the exponent (instead of the usual 8 for single or 11 for double) and 24 or 53 bits for the significand. While this on one hand disturbs the semantics of arithmetic statements and neglects the formal model of single or double precision arithmetic, it can on the other hand deliver more accurate results like in the above example for c.

This also explains why, while the INTEL processors implement the V underflow strategy in extended precision, they behave like a U implementation when the precision control is set to $t = 24$ or $t = 53$. To illustrate this, assume the precision control is set to $t = 53$ and consider the two double precision operands $a = (1 + 2^{-52}) \times 2^{-1022}$ and $b = 1.5 \times 2^{-1}$ of which the product is given by

$$a \times b = (1 + 2^{-1} + 2^{-52} + 2^{-53}) \times 2^{-1023}$$

In pure double precision semantics, this is a V$\wedge\neg$U underflow case. When the INTEL processor executes the program statement

```
double d = a * b;
```

the value $a \times b$ is first rounded to the precision of precision control (for our example $t = 53$) with extended precision exponent range, yielding

$$(1 + 2^{-1} + 2^{-51}) \times 2^{-1023} \qquad \text{inexact}$$

This rounded result is then stored to the double precision variable d without unrounding it first: denormalization without denormalization loss takes place and

$$d = (2^{-1} + 2^{-2} + 2^{-52}) \times 2^{-1022}$$

Since the store to double precision memory involves only tininess and no inexactness, the INTEL processors do not signal underflow for V$\wedge\neg$U cases when mimicing double precision. The same holds when the precision control is set to single.

2.2.2 Erroneous double rounding

More remarkable is the way in which the floating-point unit deals with certain cases which, in pure single or double precision semantics, are U underflow cases. As an example, consider the multiplication of the single precision ($t = 24$ and $[L, U] = [-126, 127]$) operands x and y where

$$\begin{aligned} x &= 1.000\ 0000\ 0000\ 0000\ 0000\ 0001 \times 2^{-25} \\ y &= 1.111\ 1111\ 1111\ 1111\ 1111\ 1111 \times 2^{-126} \end{aligned}$$

For the computation of $x \times y$ in single precision, the intermediate values `result_ext` and `result_tmp` defined above are given by

$$\begin{aligned} \texttt{result\_ext} &= 1.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111\ 1111\ 1111\ 1111\ 1111\ 1111 \times 2^{-126-24} \\ \texttt{result\_tmp} &= 1.000\ 0000\ 0000\ 0000\ 0000\ 0000 \times 2^{-126-24} \end{aligned} \tag{5}$$

Clearly, both `result_ext` and `result_tmp` are tiny. Moreover, the U conditions for underflow are satisfied since in the process of denormalization, non-zero trailing bits will be lost. Double rounding will occur if erroneously `result_tmp` is denormalized rather than `result_ext`. Denormalizing `result_ext` means

shifting the significand 24 bits to the right and adjusting the exponent accordingly, yielding $L = -126$. This unnormalized value then has to be rounded to single precision ($t = 24$), resulting in

$$\texttt{result} = 0.000\ 0000\ 0000\ 0000\ 0000\ 0001 \times 2^{-126} \qquad \text{underflow, inexact}$$

which is the correct floating-point approximation of $x \times y$ given by (5). At the same time, both the inexact and underflow exceptions should be raised to signal denormalization loss.

When the computation of $x \times y$ is carried out on an INTEL PC platform with precision control set to $t = 24$, erroneous double rounding occurs. Taking into account the hybrid format used by the INTEL when the precision control is set to $t = 24$ (or $t = 53$), the INTEL FPU first computes `result_tmp` in the extended precision register, and apparently does not unround when storing the result to a single precision variable, delivering the doubly rounded value

$$\texttt{result} = 0.000\ 0000\ 0000\ 0000\ 0000\ 0000 \times 2^{-126} \qquad \text{underflow, inexact}$$

While the inexact and underflow exceptions are appropriately raised to signal denormalization loss, the returned value is not correct. It results from rounding to even the exact halfway value obtained by denormalizing `result_tmp` rather than `result_ext`. The IEEE standard however, as it is formulated now, admits this deviation from the principle of exact rounding, for the sake of extended-based hardware platforms.

2.3 The MpIeee class library

In [10] a multiprecision, highly performant and yet fully IEEE compliant class library is discussed, with user definable precision t and base $2 \leq \beta \leq 2^{24}$. Great care has been taken to support all IEEE features (basic operations including square root and IEEE specified remainder, exactly rounded decimal-to-binary and binary-to-decimal conversions, exactly rounded conversions between floating-point formats of different precisions including the hardware precisions, round to integral value and conversions from and to integers) without performance penalty in comparison with the fastest multiprecision libraries available. Performance timings thereof can also be found in [10]. The library can be used for rather large precisions and supports an exponent range equal to the range of the C++ `long` integer type (which is currently often 32 bits), with the restriction that $|L| < U$ in (1). The implementation supports U underflow signaling by default. The fact that, generally speaking, underflow will occur less often because of the wide exponent range together with the fact that the unrounding of `result_tmp` is not so costly compared to the multiprecision basic operations, makes the cost of U underflow checking relatively small. However, if the user prefers yet more performance at the cost of reduced reliability, he or she can switch, at compile-time, to the less stringent V-, or W- or even flush-to-zero handling of underflow. The importance of the U underflow implementation in the context of a high precision library lies in the fact that a warning is only issued in case of a true potentially dangerous underflow.

References

[1] W.J. Cody, T.J. Coonen, D.M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F.N. Ris, and D. Stevenson. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE Micro*, 4:86–100, 1984.

[2] J.T. Coonen. Underflow and the denormalized numbers. *Comput. Math. Appl.*, 14:75–87, 1981.

[3] J.T. Coonen. Contributions to a proposed standard for binary floating-point arithmetic. University of California, Berkeley, 1984.

[4] A. Cuyt, B. Verdonk, and D. Verschaeren. A precision independent tool for testing floating-point arithmetic I: basic operations, square root and remainder. *ACM TOMS*, submitted.

[5] J. Demmel. Underflow and the reliability of numerical software. *SIAM J. Sci. Statist. Comput.*, 5:887–919, 1984.

[6] Floating-Point Working Group. IEEE standard for binary floating-point arithmetic. *SIGPLAN*, 22:9–25, 1987.

[7] David G. Hough et al. UCBTEST, a suite of programs for testing certain difficult cases of IEEE 754 floating-point arithmetic. Restricted public domain software from http://netlib.bell-labs.com/netlib/fp/index.html.

[8] IEEE Computer Society. IEEE standard for radix-independent floating-point arithmetic. IEEE, New York, 1987.

[9] SUN. Sun numerical computation guide, Revision A. 1995.

[10] D. Verschaeren. A class library for multiple precision IEEE floating-point arithmetic. In preparation.